

Mechanisms for Bounding Vulnerabilities of Processor Structures

Niranjan Soundararajan Angshuman Parashar Anand Sivasubramaniam

Dept. of Computer Science and Engineering

The Pennsylvania State University

University Park, PA 16802

soundara@cse.psu.edu, parashar@cse.psu.edu, anand@cse.psu.edu

ABSTRACT

Concern for the increasing susceptibility of processor structures to transient errors has led to several recent research efforts that propose architectural techniques to enhance reliability. However, real systems are typically required to satisfy hard reliability budgets, and barring expensive full-redundancy approaches, none of the proposed solutions treat any reliability budgets or bounds as hard constraints. Meeting vulnerability bounds requires monitoring vulnerabilities of processor structures and taking appropriate actions whenever these bounds are violated. This mandates treating reliability as a first-order microarchitecture design constraint, while optimizing performance as long as reliability requirements are satisfied. This paper makes three key contributions towards this goal: (i) we present a simple infrastructure to monitor and provide upper bounds on the vulnerabilities of key processor structures at cycle-level fidelity; (ii) we propose two distinct control mechanisms – throttling and selective redundancy – to proactively and/or reactively bound the vulnerabilities to any limit specified by the system designer; (iii) within this framework, we propose a novel adaptation of Out-of-Order Commit for vulnerability reduction, which automatically provides additional leverage for the control mechanisms to boost performance while remaining within the reliability budget.

Categories and Subject Descriptors

C.1.0 [Processor Architectures]: General

General Terms

Reliability, Performance

Keywords

Transient Faults, Redundant Threading, Microarchitecture

1. INTRODUCTION

With the growing need to protect processor structures from transient errors [20, 3], several recent works have proposed architectural and microarchitectural techniques to provide transient fault

tolerance for processor cores. Architecture-level approaches are attractive because of the flexibility they offer to explore a range of alternatives in the cost, performance and reliability tradeoff space. However, high-level architecture design lies at an early stage in the processor design cycle, and it is difficult to determine at this stage whether an architectural fault-tolerance mechanism will necessarily satisfy the hard reliability budgets that real systems are required to meet. Therefore, such mechanisms must either:

- (a) conservatively provide full redundancy to reduce the effective architectural vulnerability of the protected structures to zero while incurring heavy performance or implementation costs, or,
- (b) provide flexible mechanisms with controllable *knobs*, such that when the design matures and raw error rates are known, the knobs can be adjusted to ensure that the mechanism *guarantees* to satisfy any required vulnerability bound.

Previous Work: Previous (micro)architecture-level transient fault tolerance techniques have either implemented full redundancy [16, 22, 8, 11] at the cost of significant performance and/or area overheads, or attempted to provide partial redundancy *without* the ability to guarantee any vulnerability bounds [21, 5, 12, 15].

A cost-effective technique to achieve perfect fault coverage in the processor pipeline is to redundantly execute an instruction stream on two contexts of a simultaneous multithreading processor [16, 17]. Resource contention between the two threads typically leads to 20%-30% degradation in performance. Addressing this performance loss, and acknowledging that full redundancy (zero vulnerability) is not a strict requirement for most systems, recent studies [5, 12, 15] have attempted to achieve greater performance via *partial* redundancy mechanisms. Opportunistic Transient Fault Detection [5] uses spare bandwidth in the processor to decide which instructions need to be redundantly executed, with redundancy being avoided during heavy structure utilization periods. Partial redundancy reduces vulnerabilities compared to single-thread execution, but this approach cannot provide any vulnerability guarantees or bounds. Other approaches [12, 15] select instructions for redundant execution based on the value and control flow locality in the program. These approaches can achieve reasonably small vulnerabilities for processor structures, but again are unable to place bounds on these vulnerabilities for any arbitrary application. Finally, all these proposals offer one specific point of operation in the performance-reliability design space. It is not clear how much performance can be gained/lost if the system designer mandates a more relaxed/stringent reliability budget.

Our Contributions: We present and evaluate two distinct mechanisms to control vulnerabilities of processor structures and bound them under specified budgets. Both mechanisms have fixed imple-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISCA'07, June 9–13, 2007, San Diego, California, USA.

Copyright 2007 ACM 978-1-59593-706-3/07/0006 ...\$5.00.

mentation (area) costs (comparable to previous approaches), and trade off performance in unique ways to achieve as much fault tolerance as is required to meet the specified vulnerability bound.

Vulnerability control can be specified and performed for any processor structure. In this paper, we primarily apply our techniques to the Reorder Buffer (ROB), a structure that contributes significantly to the processor’s real estate. Controlling the vulnerability of one structure can have rippling effects on other structures as well. We discuss towards the end of the paper how ROB vulnerability control affects the vulnerability of the Issue Queue.

Our first contribution is in the form of an implementable technique to monitor the architectural vulnerability of the ROB online. Armed with this monitoring infrastructure, we explore two mechanisms for vulnerability control (VC) of the ROB. The first mechanism is a dispatch throttling technique, which estimates the AVF of the structure and proactively forbids an instruction from entering the structure if the action would result in violation of the specified vulnerability bound. The second control technique does not explicitly interfere with the flow of instructions, but detects vulnerability bound violations and then reactively performs redundant execution of those instructions that could have caused these violations. We also propose a hybrid mechanism that integrates these two in a complementary fashion. In addition, we analyze the effects of committing instructions out-of-order (OoOC), which can reduce the residence time of bits in the structure, thereby lowering vulnerability and providing more headroom for the VC mechanisms to improve performance.

We model and simulate these techniques on a detailed, cycle-accurate processor model, and provide simulation results for all 26 SPEC CPU2000 benchmarks. Our results show that under extremely relaxed vulnerability bounds, throttling is an attractive approach given its very low implementation overheads. For all other vulnerability bounds, selective redundancy and hybrid approaches provide significantly better performance while ensuring that the bounds are satisfied.

2. BACKGROUND AND MOTIVATION

Measuring System Vulnerability.

The vulnerability of any system component is estimated by first performing circuit-level analyses to arrive at a raw error rate (we assume a Single Event Upset error model in this work), usually expressed in terms of Mean Time Between Failures (MTBF) or Failures in Time (FIT) [9]. The raw FIT rate (FIT_{raw}) is then *derated* because microarchitectural and architectural effects reduce the probability that a transient error in the structure will actually lead to an observable error in the output. This probability can be encapsulated in terms of *Architectural Vulnerability Factors* (AVF) [9, 7]. The effective FIT rate of a structure is given by¹:

$$FIT_{eff} = AVF \times FIT_{raw} \quad (1)$$

The effective FIT rate of the entire system is obtained by accumulating the effective FIT rates of all its constituent components.

Computing Architectural Vulnerability Factors.

The AVF of a structure captures the probability that a transient error in the structure will manifest itself in observable output. At a certain point in time, any bit in a structure can be classified as

¹This work is primarily concerned with architectural and microarchitectural issues and therefore, for clarity, we assume that propagation and timing effects (PVF, TVF) are integrated into the raw FIT rate estimates.

either ACE (required for architecturally correct execution) or un-ACE. Only errors in ACE bits will result in observable errors. Un-ACE-ness arises from several sources: un-occupied structural entries, mis-speculated instructions, dynamically dead instructions, logical masking, etc. [9]. The AVF of the structure is defined as the average-over-time of the ratio of ACE bits in the structure to the total number of bits in the structure.

The AVF metric is useful to architects because it de-couples process-technology and circuit-level effects from architectural and microarchitectural effects on soft error rates, thereby enabling quantitative analysis of architectural transient fault-tolerance solutions independent of underlying variables.

Meeting Reliability Budgets.

Vendors need to ensure that their systems meet the requirements of certain *reliability budgets* specified in terms of FIT rates. System reliability budgets can be translated into individual component FIT budgets. Given a target FIT budget for a component or structure, and an estimated circuit error rate, it is possible to determine an AVF bound that an architectural transient fault-tolerance solution must satisfy in order to meet the targeted system reliability budget. However, the circuit-level estimates are usually not known during the high-level architectural design stages of the processor design cycle. Further, vendors often deploy the same microarchitecture on multiple platforms with varying configurations and different reliability budgets. This requires architectural solutions to be able to meet any arbitrary AVF bound that can be “dialed-in” when the final circuit estimates and system configuration become available. The ability of a mechanism to adjust itself to meet any specified AVF bound is sufficient to guarantee that this mechanism can be used to satisfy any arbitrary FIT budget for the component, and in turn the entire system.

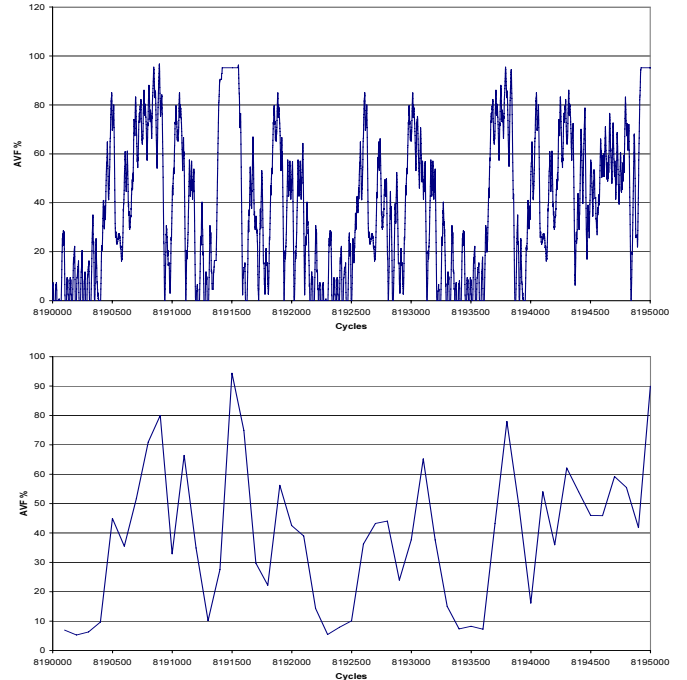


Figure 1: Dynamic AVF variation in 177.mesa at 1-cycle (left) and 100-cycle (right) granularity.

Temporal Variation of AVF.

Figure 1 shows the variation of the ROB-AVF for *177.mesa* over a period of 5000 cycles, at cycle-level and 100-cycle granularities. The plot reveals that even at the cycle-level, vulnerability varies significantly, reaching near-0% levels at times and growing as high as 95% at other instances. The *average* AVF of the benchmark is about 40%. This brings up the issue of the *time granularity* at which a certain AVF bound needs to be satisfied. Guaranteeing to meet an average AVF bound over the execution of the entire benchmark is a simpler problem than ensuring that a stringent vulnerability bound is maintained every cycle, but could result in time periods where structure vulnerabilities are extremely high, which could be unacceptable for critical systems. In this work, we design our monitoring and control mechanisms to provide vulnerability guarantees at strict cycle-level granularities, giving us worst-case performance estimates. If the bounds are relaxed, more performance could be achieved using our mechanisms.

3. DYNAMIC VULNERABILITY MONITORING

The first step in building a dynamic vulnerability control system for a processor core is to design an online AVF monitoring infrastructure. This is a non-trivial task, since near-accurate AVF estimation is an exceedingly complex process, even for offline analysis. Detection of many of the conditions that cause un-ACE-ness requires dependency chain analysis across a large sequence of instructions, which is not feasible to carry out online. There have been efforts to predictively estimate AVF by observing phased behavior of applications [4], but such statistical estimates are unsuitable for our purpose since we wish to *guarantee* that the architectural vulnerability of the structure does not exceed a given bound during any time interval.

Baseline Microarchitecture.

We assume a microarchitecture that uses a coupled Re-Order Buffer/Physical Register File, similar to that used in the Intel P6 [18]. Henceforth, we will use the term ROB to refer to this integrated structure. The Architected Register File is maintained as a separate structure into which instructions retiring from the ROB write their results. The Issue Queue stores the operand values (whenever available) along with tags for un-scheduled instructions.

Obtaining Upper Bounds on ACE-bit Counts.

Although ACE-ness/un-ACE-ness can be caused by several factors, a significant fraction of the AVF of a structure can be accounted for by tracking the residencies of non-speculative instructions in the structure, and conservatively ignoring un-ACE-ness arising from complex dependence-based factors. Using residency periods alone, it is possible to compute an *upper bound* on the AVF of a structure. For example, if 64 of the 128 entries in a processor's ROB are un-occupied in a certain cycle, then (assuming that the Head and Tail pointers are invulnerable to faults) it is guaranteed that the AVF of ROB can be at most 50% during that cycle. Next, if it is known that the results of certain instructions have not yet arrived, then this factor can be further de-rated. Finally, the contribution of wrong-path (mis-speculated) instructions to ACE-ness can also be discounted. These are all observable phenomena that can be tracked using simple monitoring logic to provide an upper bound on the ROB AVF at a cycle-accurate granularity.

Incremental Counting.

Given the number of ACE bits in the structure in any cycle, it is

possible to compute the number of ACE bits in the next cycle by observing the number of dispatches, writebacks and commits into the structure in the current cycle. If the initial number of ACE bits in the structure (when the processor is initialized) is assumed to be zero, it is possible to determine the number of ACE bits in any cycle.

We use the following notation for the remainder of this paper:

Known Parameters	
N	= No. of entries in ROB
B	= Bits in a ROB entry
R	= Bits in result field of ROB
AVF_{max}	= ROB AVF bound
Tracked Quantities	
d_i	= # dispatches in cycle i
d_i^{cp}	= # correct-path dispatches in cycle i
w_i	= # writebacks in cycle i
c_i	= # commits in cycle i

Apart from d_i^{cp} , which requires oracle knowledge or explicit back propagation of information, all of the other quantities can be easily monitored. We maintain three different estimates on the total number of vulnerable bits (TVB) in the structure: (1) TVB^{base} is the basic count of number of entries in ROB and does not account for un-ACE-ness due to un-resolved mis-speculated instructions or pending writebacks, (2) TVB^{wb} accounts for un-ACE-ness due to pending writebacks, and (3) TVB^{wb+ms} takes into account un-ACE-ness due to pending writebacks and un-resolved mis-speculated instructions in the ROB. Given the TVBs for any cycle i , the TVBs for the next cycle can be computed using the following equations:

$$\begin{aligned}
 TVB_{i+1}^{base} &= TVB_i^{base} + (d_i \times B) - (c_i \times B) \\
 TVB_{i+1}^{wb} &= TVB_i^{wb} + (d_i \times (B - R)) + (w_i \times R) - (c_i \times B) \\
 TVB_{i+1}^{wb+ms} &= TVB_i^{wb+ms} + (d_i^{cp} \times (B - R)) + (w_i \times R) - (c_i \times B)
 \end{aligned}$$

When a branch misprediction is detected, TVB^{base} and TVB^{wb} need to be re-constructed by subtracting the contribution of the mis-speculated instructions to the respective counts. This can be done in parallel with the pipeline flush and rename-table reconstruction. While TVB^{base} and TVB^{wb} can be counted online, tracking TVB^{wb+ms} is much more complex. Therefore, we use it only for illustrative purposes and do not use it with any VC mechanism. Note that if ACE_i is the actual number of ACE bits in the ROB in cycle i (as established by a hypothetical “perfect” analysis technique), then:

$$ACE_i \leq TVB_i^{wb+ms} \leq TVB_i^{wb} \leq TVB_i^{base} \quad (2)$$

Interaction with VC mechanism.

To guarantee that the AVF bound is satisfied, a VC mechanism first computes maximum number of allowable ACE bits in the structure as $ACE_{max} = AVF_{max} \times N \times B$. Then, it needs to ensure that in every cycle i , $ACE_i \leq ACE_{max}$. From Equation 2 it should be clear that for any of the TVBs, ensuring that $TVB_i \leq ACE_{max}$ will satisfy the vulnerability bound.

4. VULNERABILITY CONTROL VIA THROTTLING (VCT)

Throttling is based on the observation that it is possible to bound the AVF of a structure by conservatively controlling access to the

structure such that the number of ACE bits in the structure does not exceed ACE_{max} .

The number of ACE bits in the ROB can increase from two events: (a) dispatch, and (b) writeback. In the absence of a replay-based scheduling mechanism [6], it is difficult to control or throttle writebacks deterministically. Therefore, we employ *Dispatch Throttling* to bound the vulnerability of the ROB. Similar techniques have been used previously for power-reduction purposes [13].

Dispatch Throttling is a *proactive* VC mechanism for the ROB which stalls instruction dispatch if dispatching the next instruction into the ROB could cause TVB to exceed ACE_{max} . Since (a) writebacks cannot be throttled, and (b) writebacks affect TVB , dispatch throttling uses the strictly conservative TVB^{base} to make the decision to stall. Since TVB^{base} considers an entire occupied ROB entry as ACE and counts all speculative instructions as ACE , dispatch throttling has *exactly* the same effect as reducing the size of the ROB. Note that this is an artifact of our inaccurate estimation technique and not a critical flaw with the throttling mechanism itself. Another issue with throttling is that although it reduces the AVF of a structure over a certain period of time, it increases the execution time of the application roughly proportionately, thereby having little or no effect on the *cumulative architectural vulnerability* (the probability that the entire application execution incurs an error). However, it is extremely simple to implement and requires no logic apart from the TVB^{base} monitoring and is therefore a useful technique where AVF, and not cumulative vulnerability, is the primary concern. Our results will show later that VCT provides good performance at relaxed AVF bounds.

5. VULNERABILITY CONTROL VIA SELECTIVE REDUNDANCY (VCSR)

Selective Redundancy is based on the observation that if an instruction is redundantly executed through the processor pipeline, then the bits in a structure through which both copies of the instruction flowed during their execution are rendered effectively invulnerable. This property is also exploited by *Redundant Multithreading* techniques that fully replicate an execution thread to achieve fault tolerance [16, 14, 8, 11].

The goal of our VCSR approach is to satisfy a hard vulnerability bound at all times, while attempting to optimize performance. Though the paper talks about VCSR for the ROB, the key principles and mechanisms should be adaptable for other structures as well.

VCSR, similar to VCT, monitors the flow of ACE bits through the ROB every cycle and guarantees that the total number of ACE bits do not exceed a fixed bound. However, unlike VCT’s proactive approach, VCSR is a *reactive* technique: In any cycle, if the number of ACE bits exceed the target bound, then a subset of instructions that contributed to the ACE bits in that cycle are selected and redundantly executed, thereby effectively transforming them to un-ACE bits. VCSR’s reactive nature enables it to use TVB^{wb} and avoid a significant amount of the excess conservatism of VCT. Further, VCSR’s cost for redundancy is the over-utilization of processor resources, which has far less performance overheads than the performance reduction due to the stalls and pipeline bubbles caused by VCT, especially at low percentage-AVF bounds.

5.1 Achieving Selective Redundancy via Redundant Execution

In order to support the execution of two redundant threads in the pipeline (although one of the threads only executes a subset

of instructions), we start with a baseline Simultaneous Redundant Threading (SRT) microarchitecture [16]. The two execution streams are referred to as the *leading* (primary) and *trailing* (redundant) threads. Only non-speculative instructions are replicated in the trailing thread. Incoming loads are replicated for both threads by the Load Value Queue (LVQ). Outgoing stores from both threads are compared by the Store Checking Buffer (SCB) and only a single (checked) store is sent out to the memory system.

For VCSR, the leading thread is the primary execution thread and is executed in full. If the AVF bound of the ROB is violated in any cycle, due to excessive number of ACE bits belonging to leading thread instructions ², then a set of instructions are chosen for redundant execution via the trailing thread. Therefore, a mechanism is required that allows selective redundancy in the trailing thread.

The primary issue that must be dealt with in order to achieve selective redundancy for the trailing thread is maintaining correct architectural dataflow. Since a partial set of instructions are being executed in this thread, instructions whose producers were not executed require their source operands to be forwarded from the leading thread. Despite this forwarding, redundancy can be maintained if all instructions are checked for correctness before being committed [11, 5].

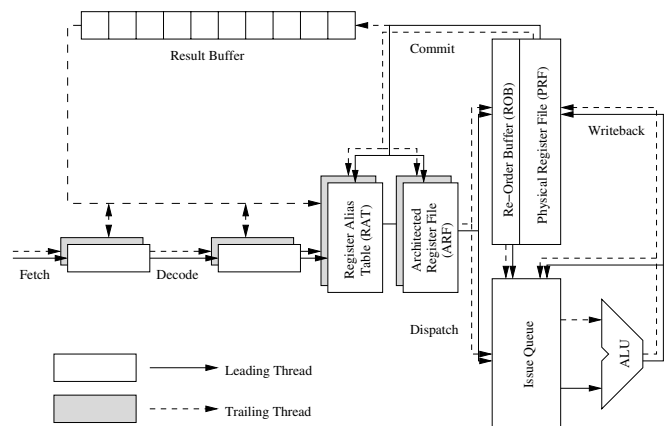


Figure 2: VCSR Pipeline Overview.

Figure 2 shows a block diagram of our microarchitecture, which is similar to that employed in [5]. Committing leading thread instructions push their results into a FIFO queue which we call the Result Buffer. The purpose of the Result Buffer is to: (a) temporarily buffer outputs until the trailing thread can compare them against its own outputs for redundancy, and (b) provide source operands for instructions that the trailing thread does not execute. The result buffer also contains a single bit that specifies whether the instruction needs to be redundantly executed or not.

5.1.1 Maintaining a Consistent Architected State

The trailing thread maintains a *consistent architected state* in its Architected Register File (ARF). Recall that the trailing thread does not mis-speculate control flow due to the presence of the Branch Outcome Queue (BOQ). In VCSR, the trailing thread fetches and pre-decodes all non-speculative instructions, in the Result Buffer, regardless of whether they are re-executed or not. If selected, the

²In a Single Event Upset model, all bits belonging to the trailing thread are un-ACE.

instruction is sent through the rest of the pipeline. Otherwise the Result Buffer entry is used to update the trailing thread's ARF. The Result Buffer entry needs to be held until the instruction commits so that the redundancy check can be performed.

Careful synchronization is needed to ensure that the register update is consistent for instructions that are not re-executed. The actual update must be performed when this instruction would have been renamed, although the instruction does not need to physically go through the decode and rename logic. This ensures that the RAT is consistent with the instruction flow. We also ensure that all trailing thread Loads, Branches and Stores perform the necessary synchronization with the LVQ, BOQ and SCB so that these structures remain consistent. Since the trailing thread's ARF is maintained in a consistent state, and stores to the system are synchronized with the trailing thread's commit point for the stores, this ARF can be defined as a precise state for establishing checkpoints, both for interrupt handling as well as error recovery.

5.2 Selecting Instructions for Redundant Execution

Given an infrastructure to dynamically monitor AVFs and a selective redundancy mechanism, we now need to come up with a policy that selects a set of instructions for redundant execution so that the AVF bounds for the ROB are met every cycle. Ideally, we would like to select the set of instructions that provides the best possible performance among all possible sets that satisfy the vulnerability criteria. While it is obvious that arriving at such an optimum solution is out of the question for a real implementation, it is also infeasible to perform any offline analysis to determine an "OPT" that can aid in evaluation of implementable heuristics. In this paper, we provide one implementable *greedy* heuristic to satisfy the constraint. We leave a comparative evaluation against other possible heuristics to future work.

VCSR heuristic can flag an instruction for redundant execution at either of the following two stages:

- **During Dispatch:** If it is determined that dispatching an instruction will cause a violation, i.e., if $(B - R) + TVB^{wb} > ACE_{max}$, then the instruction needs to be marked as requiring redundant execution. This flag is stored in the ROB until commit, upon which it is transferred to the Result Buffer.
- **During Writeback:** If a writeback is causing a violation, i.e., $R + TVB^{wb} > ACE_{max}$, then the instruction needs to be flagged for redundant execution.

The above description is a simplified view of the process. In reality, several instructions could be undergoing dispatch, writeback and commit simultaneously in a single cycle, and the flagging mechanism needs to take all of these as input to determine the set of instructions to be flagged. We give priority to younger instructions for removal since they have a greater probability of being mis-speculated instructions.

The detection and flagging logic can be kept off the critical paths by using independent ports to access the flag bit in the ROB, and allowing for an additional cycle to write the flag bit if necessary. This is not unusual since the different fields in a ROB are typically accessed via independent ports in real microprocessor implementations.

Note that an instruction selection logic could have potentially chosen from amongst any of the instructions currently residing in the ROB, and not necessarily from amongst the instructions currently undergoing dispatch or writeback. However, selecting in this manner simplifies the decision logic, maximizes selection of possibly mis-speculated instructions (see below), and allows the flag bit

to potentially share write ports and address decoders with the status bits.

Impact of Mis-Speculation.

Although TVB^{wb} does not account for un-resolved mis-speculated instructions, observe that if there are any mis-speculated instructions in the ROB, then *it is guaranteed that any instructions that are waiting for dispatch are also on the mis-speculated path*. Since mis-speculated instructions are never executed by the trailing thread, VCSR is guaranteed to never execute any dispatching instructions that were unnecessarily tagged due to the over-counting of mis-speculated instructions by TVB^{wb} .

However, it is not possible to guarantee that instructions flagged during writeback will not lead to over-conservative re-execution. To minimize this occurrence, we try to flag for re-execution the youngest among the currently writing-back instructions.

5.3 Hybrid Vulnerability Control (VCH)

A VCSR-capable infrastructure can be augmented with a dispatch throttling mechanism to combine the advantages of both approaches. Though a multitude of heuristics are possible, here we show that a simple VCH policy can help to improve the performance of a baseline VCSR system by a reasonable margin.

Recall that one of the key problems with VCT was its over-conservatism. In our hybrid approach, we perform dispatch throttling aggressively by counting only the $(B - R)$ bits for instructions in the ROB that have not yet written back and for new instructions being dispatched. Further, throttling is activated only when it is determined that all instructions being dispatched during the current cycle will end up being replayed due to an AVF bounds violation. Note that in such a scenario, while leading thread dispatch is stalled, the entire dispatch bandwidth becomes available to the trailing thread.

6. OPTIMIZATIONS

It is possible to utilize lightweight, simple-to-implement AVF-reduction techniques for certain microarchitectural structures, and some of these have been investigated by previous work [23, 5]. Most of these techniques address the fact that vulnerability arises from long residency of inactive bits in structures, and attempt to reduce such long residence periods. The vulnerability control mechanisms we have proposed are capable of exploiting the benefits offered by such AVF reduction optimizations. By reducing the effective AVF of structures, such optimizations provide more headroom for our control mechanisms to enhance performance.

In this section, we propose to reduce the AVF of the ROB by a novel adaptation of a previously-proposed concept – Out Of Order Commit (OoOC) and analyze its behavior when used in conjunction with VCT and VCSR.

6.1 Out of Order Commit

Instructions need to commit in-order out of the processor pipeline to provide the illusion of program-ordering to the programmer, and for precise handling of exceptions and interrupts. Unfortunately, it also leads to inefficiencies in the ROB since younger instructions that have completed need to wait until they reach the head of the structure.

A number of research works in the past have attempted to tackle this problem and devise solutions that allow instructions to commit out-of-order and yet maintain correctness for handling interrupts and exceptions [1]. Within the context of our vulnerability control infrastructure, OoOC is interesting because of three key reasons:

- In-Order Commit can cause significant Complete-to-Commit delays that play a large role in increasing the AVF of the ROB. Reducing this delay has the potential to reduce AVFs and thereby improve performance with the aid of a VC mechanism.
- OoOC provides performance benefits only when entries can be re-claimed through collapsing. Even if the freed entries are not reclaimed in our schemes, simply retiring instructions reduces the number of ACE bits in the ROB, thereby providing additional headroom to both VCT and VCSR to achieve higher performance. Since we avoid collapsing, we avoid having to maintain additional dispatch IDs for all instructions and having to associatively search through these IDs for any ROB access. In-Order Commit requires the retiring logic to monitor only the oldest c (commit width) entries in the ROB to determine which instructions are ready to commit. However, for OoOC, the entire ROB needs to be monitored each cycle to determine the set of instructions ready to retire. This can be accomplished by using a priority-based selection tree. The wire delay for such a logic tree is of the order of $\log(n)$ where n is the number of entries in the ROB [10].
- In VCSR, since it is the *trailing thread* that maintains the precise architected state of the machine, the leading thread can be committed out of order without having to provide any additional infrastructure to handle precise exceptions, thereby making a strong case for an implementable OoOC mechanism.

OoOC and AVF reduction.

The architected register file (ARF) of most processors are typically 8-32 entry RAM structures that are significantly simpler than the large, multi-ported, 128-entry ROB and Physical Register Files. ECC or parity protecting an ARF is relatively cost-efficient to accomplish and simplifies the implementation of several fault-tolerance mechanisms [14, 11]. Therefore, early removal of data from an unprotected ROB/PRF into a protected ARF results in AVF reduction for the ROB without affecting the reliability of any other structure.

When used with VCSR, even the ARF need not be protected. Retiring a leading thread instruction from the ROB in VCSR requires writing the result to both the leading thread's ARF as well as the Result Buffer. This act of replication implicitly provides redundancy to the data element. AVF of the ROB AVF is reduced, but neither the ARF nor the Result Buffer need to be protected.

7. RESULTS

Our experiments were conducted via execution-driven simulation using processor models that we implemented using the SimpleScalar 3.0 toolset [2]. We evaluated our techniques using all 26 applications from the SPEC CPU2000 benchmark suite. The benchmarks were compiled for the Alpha ISA, and reference input sets were used. We measured the statistics for detailed simulation of 100 million instructions after fast-forwarding to the single SimPoint [19] of each benchmark. The parameters of our baseline model are shown in Table 1.

7.1 VCT Results

Our first set of results is for VCT, and in Figure 3 we give the performance of this technique for a wide range of AVF bounds, together with the performance of a single threaded execution (which is completely vulnerable) and the baseline Simultaneous Redundant Threading (SRT) system [16] (which provides complete re-

Baseline Datapath Parameters	
Parameter	Value
Fetch/Decode/Issue/Commit Width	6
Pipeline Stages	15
Fetch Queue Size	16
Load Value Queue (LVQ) Size	128
Branch Outcome Queue (BOQ) Size	128
Store Checking Buffer (SCB)	64
Branch-Predictor	Combined Predictor with 16K-entry meta-table. 2-lev predictor with 16K-entry L1, 16K-entry L2, 14-bit history XORed with address
RAS Size	64
BTB Size	2K-entry 4-way
RUU Size	128
LSQ Size	64
Integer ALUs	4 (1-cycle latency)
Integer Multipliers/Dividers	2 (3,20)
FP ALUs	2 (2)
FP Mult./Div./Sqrt.	1 (4,12,24)

VCT/VCSR Parameters	
Parameter	Value
AVF bounds	0%, 2%, 5%, 10%, 20% 50%, 100%
Result Buffer	128-entry RAM + 384-entry FIFO

Memory System Parameters	
Parameter	Value
L1 D-Cache Ports	2
L1 D-Cache	64KB, 4-way with 32B block (2)
L1 I-Cache	64KB, 4-way with 32B block (2)
L2 Unified Cache	512 KB, 4-way with 64B line-size (12)
I-TLB	512-entries 4-way set-associative
D-TLB	1K-entries 4-way set-associative
TLB Miss-Latency	30 cycles
Memory Latency	200 cycles

Table 1: Simulation parameters. Latencies of ALUs/caches are given in parenthesis. All ALU operations are pipelined except division and square-root.

dundancy). Since no instructions will be dispatched at 0% AVF bound, the IPC of VCT with 2% AVF bound is compared with the SRT execution. At the other end, VCT with 100% AVF bound does not throttle any instruction, and is thus equivalent to single-thread execution.

We see that the simple throttling mechanism is able to provide a wide-spectrum of operating points in the performance-reliability space. This range of operation can be better understood by examining the AVF during each cycle of the execution. Figure 4 plots the Cumulative Density Function (CDF) of the AVF during each cycle for five representative applications (others are omitted for clarity) in the single threaded execution. Based on these results, we make the following observations:

- This mechanism is effective only at high AVF bounds (typically above 50%). On the average, there is only a 9.3% drop in IPC when going to a 50% bound from a 100% bound, while there is a 83% drop when going down all the way to 2%. At high AVF values, there is no significant impact from slightly under-utilizing the ROB. This is particularly true in applications where the ROB is anyway not operating at full capacity. For instance, in *200.sixtrack*, where the ROB occupancy in the single-thread execution is 51.6% on the average, there is very little change in IPC going from 50% to 100% AVF. The low occupancy of the ROB is also reflected in the AVF CDF graph for *200.sixtrack* (where the number of cycles with AVF greater than 50% is quite low), as is the case for *181.mcf* and *189.lucas*.

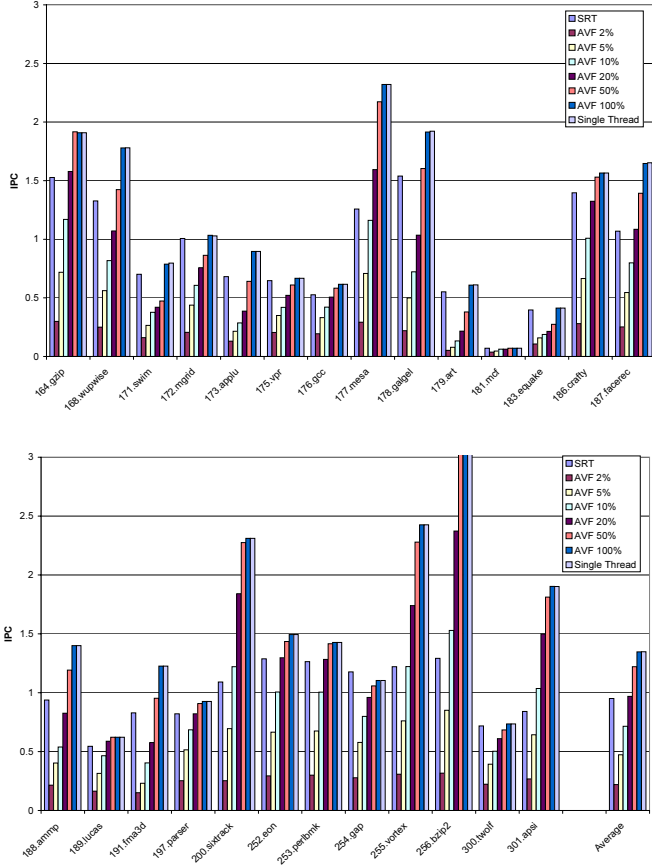


Figure 3: IPC with VCT for various AVF bounds. Also provided for each benchmark are IPCs for single-thread execution (rightmost bar) and SRT (leftmost bar).

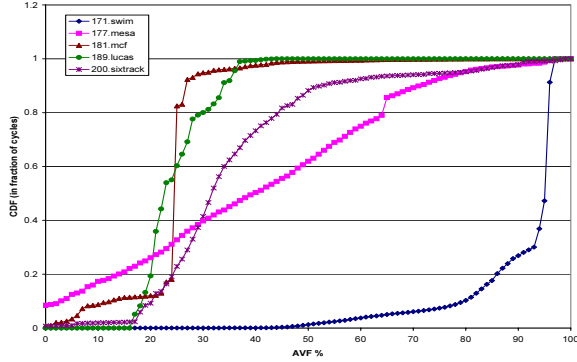
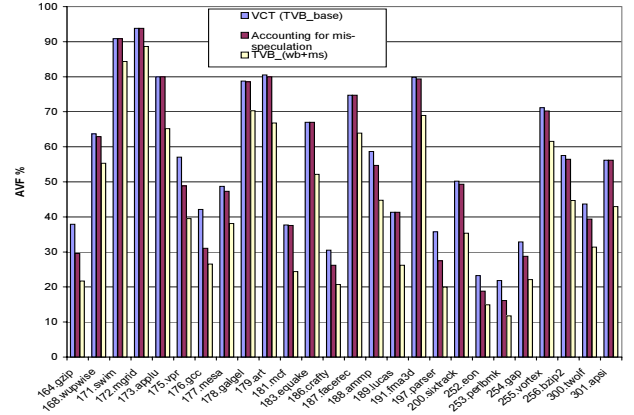


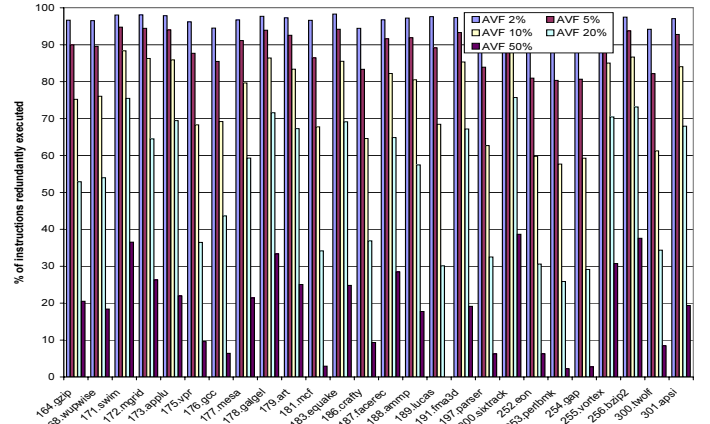
Figure 4: CDF of the cycle granularity AVF over the single-thread execution.

- At the other end, going down to a small AVF bound severely throttles the pipeline making it a much inferior alternative to a mechanism such as SRT. The throttling penalty is particularly acute for high IPC applications (e.g. *256.bzip2*, *177.mesa*). On the average, an AVF bound of 20% or higher needs to be tolerated in order for VCT to become competitive with SRT performance-wise.
- The steepness of the IPC improvement with increasing AVF bounds can be explained with the AVF graphs shown in Fig-

ure 4. (i) In applications such as *177.mesa*, the CDF graph shows that the AVF of the ROB in each cycle is more or less evenly distributed between 0-60%, implying that a 40% AVF bound will throttle half as much as a 20% AVF bound. This effect can be seen in the steady IPC improvement when going from 2% to 50% AVF bound. (ii) In applications such as *181.mcf* and *189.lucas*, the AVF CDF graph shows high convexity, i.e., most cycles have AVF less than 40%. Consequently, the steepest performance losses in VCT is incurred for AVF bounds smaller than 20%. (iii) At the other end, *171.swim* is an example application with a concave CDF graph, where most cycles have very high AVF values (close to 100%). The resulting IPC graph shows that there is a significant gap in performance between 50 and 100% AVF bounds for this application.



(a) AVF Estimation of the ROB through VCT (using TVB^{base} bits) averaged over all the cycles. Also shown are the AVF calculated by discounting mis-speculated instructions, and the AVF obtained if we had perfect Oracle knowledge using TVB^{wb+ms} bits.



(b) Percentage of instructions redundantly executed with VCSR for different AVF bounds.

Figure 5: Additional Statistics for VCT and VCSR Executions.

Another factor which makes VCT a less attractive vulnerability control mechanism is the possible overestimation of vulnerable bits of the ROB in any cycle. We quantify this conservative estimation of vulnerable bits by VCT in Figure 5 (a), by comparing its average AVF with that of an execution (i) which accounts for mis-

speculated instructions, and (ii) which accounts for output result bits when they become available as well as mis-speculated instructions. Across the applications, there is an absolute error of around 10-20% in the AVF estimation by VCT (most of it contributed by early accounting of writebacks). While intuitively it could seem that the performance degradation of VCT will be more prominent in applications with high AVFs, the error in its estimation can cause VCT to perform poorly even in applications with low AVFs. For instance, even though *253.perlbnk* has an average AVF of close to 10%, we see that a 20% AVF bound VCT execution performs poorly because the conservatism puts the estimation over 20% on the average. Applications with lower AVFs are thus hurt more by the over-estimation in VCT.

7.2 VCSR Results

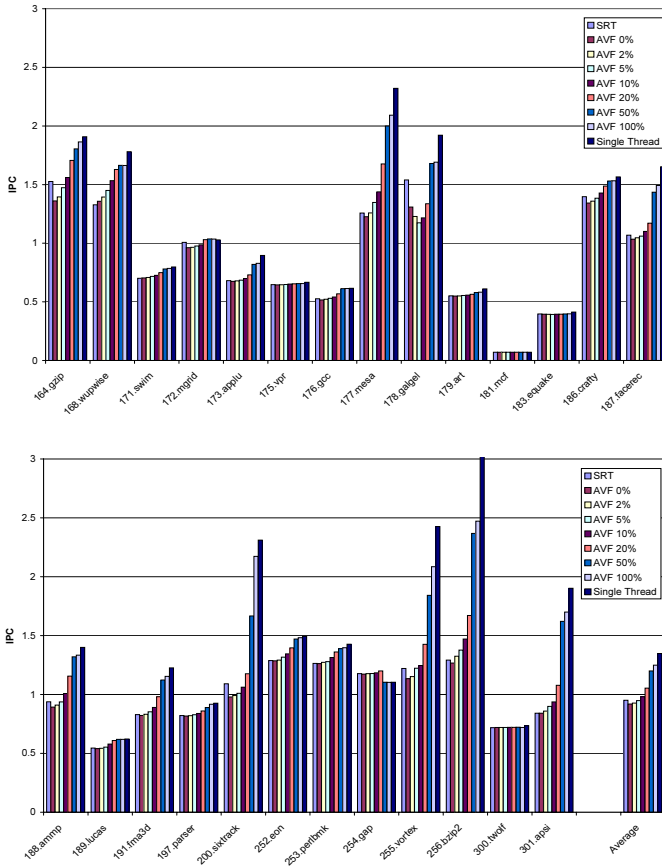


Figure 6: IPC with VCSR for various AVF bounds. Also provided for each benchmark are IPCs for single-thread execution (rightmost bar) and SRT (leftmost bar).

Figure 6 shows the VCSR results for different AVF bounds, in comparison with SRT and single-thread executions. A good VCSR implementation should perform close to single-threaded execution for a 100% AVF bound (no instructions are redundantly executed), and close to SRT for a 0% AVF bound (all instructions are selected for re-execution). In these two scenarios, our VCSR implementation suffers a performance degradation of 7% and 3% respectively, which we consider to be fairly efficient.

As in the earlier set of experiments, lowering the AVF bound reduces the IPC of the execution. In VCSR, this is because lowering AVF bounds leads to more instructions getting picked for redundant

execution as is shown in Figure 5 (b). However, the steepness of the decline in IPC for low AVF bounds is lesser compared to VCT. This can be explained by two main advantages that VCSR has over VCT. First, since VCSR is reactive it is much less conservative than VCT which is pro-active and hence takes appropriate actions only when the AVF has *actually* exceeded the bound, compared to VCT which takes pro-active actions based on predicted (over-estimated) AVF values. Second, stalling based control (VCT) can have more detrimental impact on pipeline performance (reducing parallelism, introducing bubbles, extending critical paths, etc.), compared to higher resource contention between competing threads (VCSR). The latter artifact is particularly apparent in memory bound applications such as *171.swim* and *189.lucas*, where memory stalls in the leading thread can cause less contention for datapath resources between the two threads. Consequently, even when over 95% of the instructions are redundantly executed, the performance loss compared to single-thread execution is only around 5% in these applications. This is also the reason why SRT performance in these applications is not significantly worse than single-thread execution either.

Resource contention in VCSR is expected to play a more detrimental role in applications with high IPC. For instance, applications such as *256.bzip2*, *255.vortex* and *200.sixtrack*, show significant drop in IPC even when going to AVFs of 20% with VCSR. In some of these cases, VCSR actually does worse than VCT for high AVF bounds. For instance, *200.sixtrack* with a AVF bound of 50% does 27% worse in VCSR compared to VCT. The resource contention in these high ILP applications is more detrimental, and throttling to slightly lower the effective ROB size is a more effective option to keep the AVF under control when the bounds are lenient.

7.3 VCH Results

Summarizing the results from the previous two sets of experiments, we see that (i) pro-active throttling (VCT) hurts when the AVF bounds are stringent, and (ii) reactive selective replication (VCSR) causes resource contention for high ILP applications at high AVF bounds and incurs implementation overheads (i.e., at 100% AVF bound, the performance is not the same as single-thread). We illustrate these observations in Figure 7. This graph plots the performance normalized with respect to single-thread execution for different AVF bounds. Rather than show this for each application, the graph has been drawn by taking the geometric mean of the slow-downs observed by all 26 applications. As we can see, the curve for VCT depicts worse performance at stringent AVF bounds, but crosses over the curve for VCSR for AVF bounds greater than 75%.

Of these two, VCSR is a better option because it offers meaningful operating points for a wide range of useful AVF bounds, including a 0% AVF bound. However, it is possible to integrate these two mechanisms to co-exist, as was explained earlier in section 5.3. The line, VCH, in Figure 7 plots the performance of this hybrid mechanism. We see that VCH does better than the other two for a wide spectrum of specified AVF bounds. Even for an AVF bound of 2% it brings the performance within 18% of single thread IPC on the average, and it is 10% better than SRT.

7.4 OoO Commit

Our final set of results are intended to show how committing instructions OoO to reduce ROB vulnerable bits can help our mechanisms boost performance. In the interest of clarity, we show this for only the VCSR mechanism in Figure 9. Note that these results are with a non-collapsing ROB, where the vacated entries are left unutilized until they reach the head of the ROB. Consequently, SRT

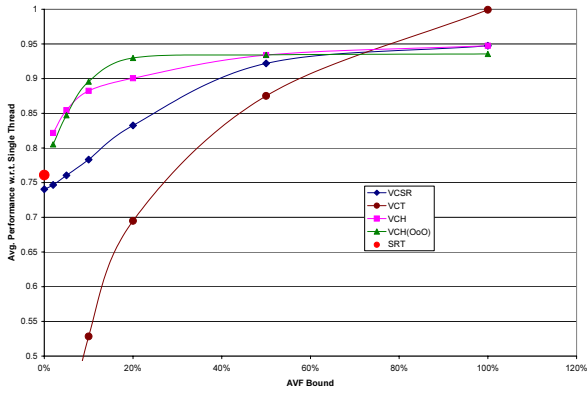


Figure 7: Summary of Results. Geometric Mean of normalized IPC (w.r.t. single thread IPC) across applications for different AVF bounds. Also shown is SRT performance on the Y-axis (AVF = 0).

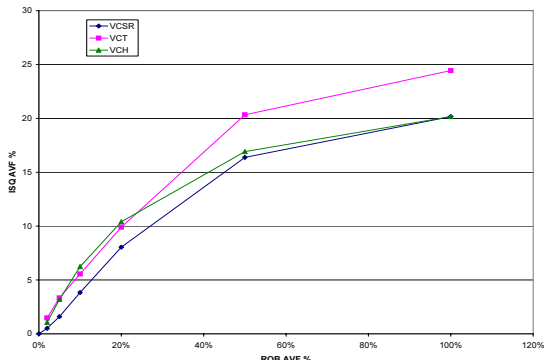


Figure 8: Effect of Controlling the AVF of the ROB on the AVF of the Issue Queue.

and single-thread performance is not going to be any different from that shown earlier in Figure 6, and is not repeated in these graphs.

By reducing residency times of instructions in the ROB, the VCSR control mechanism finds greater vulnerability slack, and needs to step in less often to mark instructions for redundant execution. The consequent reduced number of redundant instructions automatically boosts the performance as we can see in these results. This improvement is particularly noticeable in high IPC applications where resource contention due to redundant execution with a low AVF bound was hurting performance previously. For instance, when we consider *200.sixtrack*, we now see that the IPCs with AVF bounds of 20% and 50% matching those for a bound of 100%, while the IPC of the 20% bound was half as much in the in-order execution. We have also conducted experiments for OoO commit with VCH, and the resulting performance is summarized in the line shown as “VCH(OoO)” in Figure 7.

7.5 Impact of VC on other structures

This paper has focused on controlling the vulnerability of the ROB. However, in order to meet the overall system reliability budget, it is likely that the vulnerability of multiple processor structures will need to be controlled. This is a multi-dimensional optimization problem that we intend to explore in future work. This is especially interesting because of the fact that controlling the vulnerability of one structure could impact the vulnerability of other structures. In general, redundant execution benefits all structures in the pipeline. Therefore, reduction in vulnerability of the ROB will lead to reduction in vulnerability of other structures such as the Issue Queue and

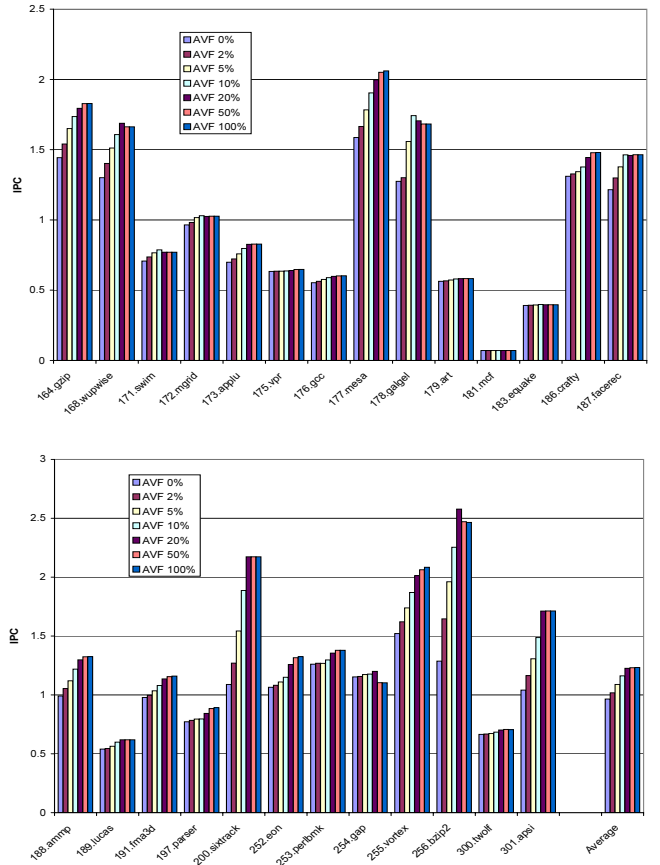


Figure 9: IPC with VCSR using Non-collapsing OoO Commit.

Load-Store Queue as well. Consequently, VCSR based approaches should lower the AVF on the Issue Queue better than VCT. We provide some preliminary results that show the observed impact of bounding the ROB’s AVF (shown on the x-axis) on the AVF of the Issue Queue in Figure 8.

8. CONCLUSION

This paper has presented knobs for controlling the vulnerability of processor structures that can be modulated to meet target reliability budgets specified by system designers. We have proposed two distinct mechanisms to control the vulnerability of the ROB of a processor, aided by a simple online monitoring infrastructure which provides online upper-bound estimates on the AVF of the ROB at a cycle-level fidelity. The control mechanisms operate by proactively (throttling) and/or reactively (re-execution) ensuring that the number of vulnerable bits in the ROB do not exceed a specified threshold. Using detailed simulations we have evaluated the pros and cons of our approaches.

Our results show that throttling suffers significant performance impacts under stringent bounds, but its low overheads help it to perform better than selective redundancy under relaxed (high) vulnerability bounds. It is also simple to implement. Selective redundancy can provide operating points that span the entire performance spectrum between complete redundancy and single thread execution while meeting any specified budget. We have also explored the benefits of integrating these two mechanisms.

Finally, we have proposed a novel adoption of Out-of-Order instruction commit for vulnerability reduction. Avoiding the recla-

mation of holes left by committed instructions simplifies our implementation while reaping the benefits of vulnerability reduction. Further, with selective redundancy, the trailing thread can be used to maintain precise machine state, removing the need to explicitly handle this issue for out of order commit. The vulnerability reduction provided by OoOC provides our control mechanisms with more AVF headroom to boost performance.

Our future work involves investigating the issue of controlling vulnerability of multiple structures in a coordinated manner. We are also exploring relaxed specifications of vulnerability bounds, which need to be adhered to only with a high percentile as opposed to our current deterministic cycle-granularity bounds.

Acknowledgements

We would like to thank the anonymous reviewers for their detailed comments which helped us improve the quality of the presentation. This research was funded in part by NSF grants 0103583, 0509234, 0325056 and 0429500.

9. REFERENCES

- [1] G. B. Bell and M. H. Lipasti. Deconstructing commit. In *Proceedings of the 4th International Symposium on Performance Analysis of Systems and Software*, Austin, Texas, March 2004.
- [2] D. Burger and T. Austin. The SimpleScalar Toolset, Version 3.0. <http://www.simplescalar.com>.
- [3] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4):14–19, July-August 2003.
- [4] X. Fu, J. Poe, T. Li, and J. A. B. Fortes. Characterizing microarchitecture soft error vulnerability phase behavior. In *MASCOTS '06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 147–155, Washington, DC, USA, 2006. IEEE Computer Society.
- [5] M. A. Goma and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 172–183, 2005.
- [6] I. Kim and M. H. Lipasti. Understanding scheduling replay schemes. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 198, Washington, DC, USA, 2004. IEEE Computer Society.
- [7] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Softarch: An architecture level tool for modeling and analyzing soft errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 496–505, 2005.
- [8] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 99–110, May 2002.
- [9] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 29–40, December 2003.
- [10] S. Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin - Madison, 1998.
- [11] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 376–386, June 2004.
- [12] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. Slick: Slice-based locality exploitation for efficient redundant multithreading. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.
- [13] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 90–101, Washington, DC, USA, 2001. IEEE Computer Society.
- [14] J. Ray, J. Hoe, and B. Falsafi. Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 214–224, December 2001.
- [15] V. K. Reddy, S. Parthasarathy, and E. Rotenberg. Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.
- [16] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 25–36, June 2000.
- [17] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*, pages 84–91, June 1999.
- [18] J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors (Beta Edition)*. McGraw Hill, 2003.
- [19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.
- [20] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2002.
- [21] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: improving both performance and fault tolerance. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 257–268, 2000.
- [22] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 87–98, May 2002.
- [23] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt. Techniques to Reduce the Soft Error Rate of High-Performance Microprocessor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 264–275, June 2004.